Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 56 (2005) 251–273
Science of Computer Programming
www.elsevier.com/locate/scico

Model-checking processes with data

J.F. Groote*, T.A.C. Willemse

Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands

Received 1 August 2003; received in revised form 8 July 2004; accepted 3 August 2004
Available online 8 October 2004

Abstract

We propose a procedure for automatically verifying properties (expressed in an extension of the modal μ-calculus) over processes with data, specified in μCRL. We first briefly review existing work, such as the theory of μCRL and we discuss the logic, called *first order modal μ-calculus* in more detail. Then, we introduce the formalism of *first order boolean equation systems* and focus on several lemmata that are at the basis of the soundness of our decision procedure. We discuss our findings on three non-trivial applications for a prototype implementation of this procedure. The results show that our prototype can deal with quite complex and interesting properties and systems, showing the efficacy of the approach.

Keywords: Model checking; Verification; Process algebra; μCRL; Modal μ-calculus; Boolean equation systems; Infinite state systems

1. Introduction

Model checking has come about as one of the major advances in automated verification of systems in the last decade. It has earned its medals in many application areas (e.g. communications protocols, timed systems and hybrid systems), originating from both academic and industrial environments.

* Corresponding author.
E-mail addresses: j.f.groote@tue.nl (J.F. Groote), t.a.c.willemse@tue.nl (T.A.C. Willemse).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.08.002

However, the class of systems to which model checking techniques are applicable is restricted to systems in which dependencies on infinite datatypes are absent, or can be abstracted from using dedicated techniques. Examples of such dedicated techniques are the use of e.g. regular expressions [1] and queue representations [5] for communications protocols, Presburger arithmetic [11] for networks and counter abstraction [25] for parameterised systems. While great progress has been made in these areas, they all focus on special data structures. In contrast, the approach we outline in this paper is capable of dealing with arbitrary data and process structures. Thus, we do not restrict ourselves to special classes of systems a priori.

We explore the possibility of extending model checking techniques to dealing with processes which can depend on arbitrary data types. We describe a procedure, for which we have also implemented a prototype, that verifies a given property on a given data-dependent process. The problem in general is easily shown to be undecidable, so, while we can guarantee soundness of our procedure, we cannot guarantee its termination. However, as several examples suggest, many interesting systems with infinite state spaces can be verified using our procedure. Naturally, our technique also applies to systems with finite (but extremely large) state spaces.

The framework we use for describing the behaviour of a system is process algebraic. We use the process algebraic language μCRL [13,15], which is an extension of ACP [3]; this language includes a formal treatment of data, as well as an operational and axiomatic semantics of process terms. Compared to CCS or ACP, the language μCRL is more expressive [20]. For our model checking procedure, we assume that the processes are written in a special format, the *Linear Process Equation* (LPE) format, which is discussed in e.g. [29]. Note that this does not pose a restriction on the set of processes that can be modelled using μCRL, as all sensible process descriptions can be transformed to this format [29]. When dealing with datatypes, an explicit representation of the entire state space is often not possible, since it can very well be infinite. Using the LPE format has the advantage of working with a finite representation of the (possibly infinite) state space.

The language we use to denote our properties in is an extension of the modal μ-calculus [18]. In particular, we allow first order logic predicates and parameterised fixpoint variables in our properties. These extensions, which are also described in e.g. [12], are needed to express properties about data.

The approach we follow is inspired by the work of e.g. Mader [21], and uses (in our case, first order) boolean equation systems as an intermediate formalism. We present a translation of first order modal μ-calculus expressions to first order boolean equation systems in the presence of a fixed Linear Process Equation, which is based on the translation given in [12]. The procedure for solving the first order boolean equation systems is based on the Gauß elimination algorithm described in, e.g. [21]. Remark that even though some of the theory we present in this paper can be found in slightly different settings, until now, it was very clear that this approach could be effective and feasible for automatic model checking of data-dependent systems.

This paper is structured as follows: we start by reviewing existing theory, which is done for the sake of readability. Section 2 briefly introduces the language μCRL and the Linear Process Equations format that is used in all subsequent sections. In Section 3, we describe the first order modal μ-calculus in detail, including a number of examples, illustrating the

use of the language. Section 4 discusses first order boolean equation systems and describes the translation of first order modal μ-calculus formulae, given a Linear Process Equation, to a sequence of first order boolean equations. We then propose a procedure for solving the first order boolean equations, which we describe in Section 5; its implementation is discussed in Section 6, and three sample verifications are described in Section 7. Section 8 is reserved for concluding remarks and an overview of related work.

2. The theory of μCRL

Our main focus in this paper is on processes with data. As a framework, we use the process algebra μCRL [13]. Its basic constructs are along the lines of ACP [3] and CCS [24], though its syntax is influenced mainly by ACP. In the process algebra μCRL, data is an integral part of the language, which makes the language more expressive than CCS or ACP (see discussion in [20]). As we enforce no restrictions on data or on data-types, we here introduce the more abstract notion of data by considering only a *data algebra*.

Definition 1 (Data Algebra). A *Data Algebra* is a tuple $\mathcal{A} = (\mathcal{F}, \mathcal{D})$, where $\mathcal{D}$ is a collection of sets called *data domains*. The set $\mathcal{F}$ contains functions from data domains to some single data domain.

For the exhibition of the remainder of the theory, we assume we work in the context of a data algebra without explicitly mentioning its constituent components. As a convention, we assume the data algebra contains all the required data types; in particular, we always have the domain $\mathbb{B}$ of booleans with functions $\top:\to\mathbb{B}$ and $\bot:\to\mathbb{B}$, representing *true* and *false* at our disposal.

The language μCRL has only a small number of carefully chosen operators and primitives. Processes are the main objects in the language. A set of parameterised actions *Act* is assumed; actions can be considered as functions from a data domain to a process. An action $a \in Act$ represents an atomic event, taking a number of data arguments. The process representing no behaviour, i.e. the process that cannot perform any actions is denoted δ. This constant is often referred to as *deadlock* or *inaction*. Note that all actions a terminate successfully immediately after executing the action, whereas the process $a \cdot \delta$ does not terminate successfully.

Processes are constructed using several operators. The main operators are alternative composition ($p + q$ for some processes p and q) and sequential composition ($p \cdot q$ for some processes p and q). The sequential composition operator is often not written down explicitly. Conditional behaviour is denoted using a ternary operator (we write $p \lhd b \rhd q$ when we mean process p if b holds and else process q). The process $b:\to p$ serves as a shorthand for the process $p \lhd b \rhd \delta$, which represents the process p under the premise that b holds. Recursive behaviour is specified using equations. Data is intertwined with processes such that process variables can be considered as functions from a data domain to processes. Consider the following process.

$$X(n{:}\mathbb{N}) = up \cdot X(n+1) + show(n) \cdot X(n) + [n > 0]{:}\to down \cdot X(n-1).$$

The behaviour denoted by process $X(n)$ is the increasing and the decreasing of an internal counter n or showing its current value. Note that the *up* and *down* actions do not have parameters. For the formal exposition, however, it can be more convenient to assume that actions and processes have a single parameter. This assumption is easily justified, as we can assume the existence of a singleton data domain, together with adequate pairing and projection functions.

A more complex notion of process composition consists of the parallel composition of processes (we write $p \| q$ to denote the process p parallel to the process q). Synchronisation is achieved using a separate communication function γ, prescribing the result of a communication of two actions (e.g. $\gamma(a, b) = c$ denotes the communication between actions a and b, resulting in action c). Two parameterised actions $a(n)$ and $b(n')$ can communicate to action $c(n'')$ only if the communication between actions a and b results in action c (i.e. $\gamma(a, b) = c$) and $n'' = n' = n$.

The communication function is used to specify *when* communication is possible; this, however, does not mean communication is enforced. To this end, we must encapsulate the individual occurrences of the actions that participate in the communication, such that these cannot autonomously take place. This is done using the encapsulation operator (we write $\partial_H(p)$ to specify that all actions in the set of actions H are to be encapsulated in process p).

The last operator considered here is data-dependent alternative quantification (we write $\sum_{d:D} p$ to denote the alternatives of process p, dependent on some arbitrary datum d selected from the (possibly infinite) data domain D). The $\sum$-operator is best compared to e.g. input prefixing, but is more expressive (see e.g. [20]). As an example of the $\sum$-operator we consider a process that can set an internal counter to an arbitrary value, which can be read at will:

$$V(n{:}\mathbb{N}) = \mathit{read}(n) \cdot V(n) + \sum_{n':\mathbb{N}} \mathit{set}(n') \cdot V(n').$$

For verification or analysis purposes, it is often most convenient to eliminate parallelism in favour of sequential composition and (quantified) alternative composition. A behaviour of a process can then be denoted as a state-vector of typed variables, accompanied by a set of condition–action–effect rules. Processes denoted in this fashion are called *Linear Process Equations*.

Definition 2 (Linear Process Equations). A *Linear Process Equation* (LPE) is a parameterised equation taking the form

$$X(d{:}D) = \sum_{i:I} \sum_{e_i:D_i} [c_i(d, e_i)] :\rightarrow a_i(f_i(d, e_i)) \cdot X(g_i(d, e_i))$$

where I is a finite index set; D and D_i are data domains; d and e_i are data variables; $a_i \in Act$ are actions with parameters of sort D_{a_i}; $f_i{:}D \times D_i \rightarrow D_{a_i}$, $g_i{:}D \times D_i \rightarrow D$ and $c_i{:}D \times D_i \rightarrow \mathbb{B}$ are functions. The function f_i yields, on the basis of the current state d and the bound variable e_i, the parameter for an action a_i; the "next-state" is encoded in the function g_i, and is determined on the basis of the current state and the bound variable e_i. The function c_i describes when action a_i can be executed. The data domain D is referred to as the *parameter set* of X.

In this paper, we restrict ourselves to the use of non-terminating processes, i.e. we do not consider processes that, apart from executing an infinite number of actions, also have the possibility to perform a finite number of actions and then terminate successfully. Including termination in our theory does not pose any theoretical challenges, but is omitted in our exposition for brevity. Several techniques and tools exist to translate a guarded μCRL process to linear form (see e.g. [14,29]). In the remainder of this paper, we use the LPE-notation as a vehicle for our exposition of the theory and practice.

The operational semantics for μCRL can be found in e.g. [13,15]. Since we restrict our discussions to process expressions in LPE-form, we here only provide a definition of the labelled transition system as it is induced by a process in LPE-form.

Definition 3 (Transition System of an LPE). The *labelled transition system* of a Linear Process Equation as defined in Definition 2 is a quadruple $M = \langle S, \Sigma, \longrightarrow, s_0 \rangle$, where

- $S = \{d \mid d{\in}D\}$ is the (possibly infinite) set of *states*;
- $\Sigma = \{a_i(d_{a_i}) \mid i{\in}I \wedge a_i{\in}Act \wedge d_{a_i}{\in}D_{a_i}\}$ is the (possibly infinite) set of labels;
- $\rightarrow = \{(d, a_i(d'_a), d') \mid i{\in}I \wedge a_i{\in}Act \wedge \exists_{e_i \in D_i} c_i(d, e_i) \wedge d'_a = f_i(d, e_i) \wedge d' = g_i(d, e_i)\}$ is the *transition relation*. For an LPE X, we write $X(d) \xrightarrow{a(e)} X(d')$ rather than $(d, a(e), d'){\in}\rightarrow$;
- $s_0 = d_0{\in}S$, for a given $d_0{\in}D$, is the *initial state*.

3. First order modal μ-calculus

The logic we consider is described in [12]. It is based on the standard modal μ-calculus [18], and extends it with data variables, quantifiers and parameterised fixpoints. This logic allows us to express data dependent properties. This logic is referred to as the *first order modal μ-calculus*, but, for brevity, we often write *μ-calculus*. In this section, we review its syntax and semantics, and we illustrate these by means of several small examples.

Definition 4 (Syntax of μ-Calculus Formulae). Expressions φ in the μ-calculus are defined by the following grammar:

$$\begin{array}{lcl} \varphi & ::= & b \mid Z(e) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid [\alpha]\varphi_1 \mid \forall d{:}D.\varphi \mid (\nu Z(d{:}D).\varphi)(e) \\ \alpha & ::= & a(e) \mid \top \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2 \mid \forall d{:}D.\alpha_1 \end{array}$$

where b is a boolean expression of domain $\mathbb{B}$, possibly containing data variables d of the set $\mathcal{D}$, e is a data expression (possibly containing data variables d of the set $\mathcal{D}$) of type D; Z is a propositional variable from a set $\mathcal{P}$, and $(\nu Z(d{:}D).\varphi)(e)$ is subject to the restriction that any free occurrence of Z in φ must be within the scope of an even number of negation symbols. Furthermore, α represents an *action formula*, where a is a parameterised action of set *Act*.

We restrict ourselves to μ-calculus formulae given in *Positive Normal Form* (PNF). This means that negation only occurs on the level of atomic propositions and, in addition, all bound variables are distinct. We often write σ to denote an arbitrary fixpoint, i.e. $\sigma{\in}\{\mu, \nu\}$. Note that not every μ-calculus formula can be converted into PNF.

The semantics of μ-calculus formulae is defined by means of an interpretation over a labelled transition system M that is induced by an LPE (recall Definition 3). Since μ-calculus expressions are open terms, the semantics is defined in the context of *environments*. Environments are (partial) mappings of a set of variables to elements of a given type.

We use the following convention: for a (countable) set $\mathcal{V}$ of variables, a domain of values V and an environment $\theta:\mathcal{V}\to V$, a variable $v\in\mathcal{V}$ and a value $w\in V$, we write $\theta[w/v]$ for the environment θ', defined as $\theta'(v') = \theta(v')$ for all variables v' different from v and $\theta'(v) = w$. In effect, $\theta[w/v]$ stands for the environment θ where the variable v has value w. The interpretation of a variable v in an environment θ is written as $\theta(v)$.

Definition 5 (Semantics of μ-Calculus Formulae). Let $\varepsilon:\mathcal{D}\to D$ be a data environment, and $\rho:\mathcal{P}\to(D\to 2^S)$ be a propositional environment. Let X be an LPE with parameter set E, action set Act with actions carrying parameters from parameter sets E_a for each $a\in Act$. The interpretation of a μ-calculus formula φ, denoted by $[\![\varphi]\!]\rho\varepsilon$, is a subset of S and is defined inductively as:

$$
\begin{aligned}
[\![b]\!]\rho\varepsilon &= \begin{cases} S \text{ if } [\![b]\!]\varepsilon \\ \emptyset \text{ otherwise} \end{cases} \\
[\![Z(e)]\!]\rho\varepsilon &= \rho(Z)([\![e]\!]\varepsilon) \\
[\![\neg\varphi]\!]\rho\varepsilon &= S \setminus [\![\varphi]\!]\rho\varepsilon \\
[\![\varphi_1 \wedge \varphi_2]\!]\rho\varepsilon &= [\![\varphi_1]\!]\rho\varepsilon \cap [\![\varphi_2]\!]\rho\varepsilon \\
[\![[\alpha]\varphi]\!]\rho\varepsilon &= \{w\in S \mid \forall_{w'\in S}\ \forall_{a\in Act}\ \forall_{w_a\in E_a} \\
&\qquad (X(w) \xrightarrow{a(w_a)} X(w') \wedge a(w_a)\in[\![\alpha]\!]\varepsilon) \Rightarrow w'\in[\![\varphi]\!]\rho\varepsilon\} \\
[\![\forall d{:}D.\varphi]\!]\rho\varepsilon &= \bigcap_{v'\in D} [\![\varphi]\!]\rho(\varepsilon[v'/d]) \\
[\![(\nu Z(d{:}D).\varphi)(e)]\!]\rho\varepsilon &= (\nu \Phi_{\rho\varepsilon})([\![e]\!]\varepsilon)
\end{aligned}
$$

where we define $\Phi_{\rho\varepsilon} \stackrel{\text{def}}{=} \lambda F{:}D\to 2^S.\lambda v{:}D.[\![\varphi]\!](\rho[F/Z])(\varepsilon[v/d])$. The interpretation of action formulae α, denoted $[\![\alpha]\!]\varepsilon$ is defined inductively as:

$$
\begin{aligned}
[\![\top]\!]\varepsilon &= \Sigma \\
[\![a(e)]\!]\varepsilon &= \{a([\![e]\!]\varepsilon)\} \\
[\![\neg\alpha]\!]\varepsilon &= \Sigma \setminus [\![\alpha]\!]\varepsilon \\
[\![\alpha_1 \wedge \alpha_2]\!]\varepsilon &= [\![\alpha_1]\!]\varepsilon \cap [\![\alpha_2]\!]\varepsilon \\
[\![\forall d{:}D.\alpha]\!]\varepsilon &= \bigcap_{v\in D} [\![\alpha]\!]\varepsilon[v/d].
\end{aligned}
$$

The set of functions from D to subsets of S is denoted by the $[D\to 2^S]$. On this set, we define the ordering $\sqsubseteq$ as $\varphi \sqsubseteq \psi$ iff for all $d{:}D$, we have $\varphi(d) \subseteq \psi(d)$. The interpretation of fixpoint expressions is then justified by the fact that the underlying lattice $([D\to 2^S], \sqsubseteq)$ is a complete lattice and the functionals are monotonic over this lattice, see [12]. From Tarski's Theorem [28], the existence and uniqueness of fixpoints over this lattice readily follows.

For ease of use, we introduce the following abbreviations for μ-calculus formulae φ, action formulae α and (both μ-calculus formulae and action formulae) ψ.

$$\begin{array}{ll}
\bot & \stackrel{\text{def}}{=} \neg\top \\
(\psi_1 \vee \psi_2) & \stackrel{\text{def}}{=} \neg(\neg\psi_1 \wedge \neg\psi_2) \\
\langle\alpha\rangle\varphi & \stackrel{\text{def}}{=} \neg[\alpha]\neg\varphi \\
(\exists d{:}D.\psi) & \stackrel{\text{def}}{=} (\neg\forall d{:}D.\neg\psi) \\
(\mu Z(d{:}D).\varphi)(e) & \stackrel{\text{def}}{=} (\neg\nu Z(d{:}D).\neg\varphi[\neg Z/Z])(e).
\end{array}$$

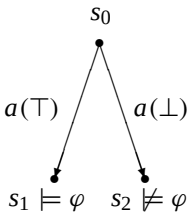
One may be led to believe that the universal quantifier in action formulae always yields the empty set. However, using negation in combination with the universal quantifier, we can obtain more exciting sets than the empty set. Below, we provide several examples to illustrate various constructs in the μ-calculus.

Example 6. Standard constructions in temporal logic often involve expressions such as "always φ" and "eventually φ". Their μ-calculus counterparts are expressed as $\nu Z.([\top]Z \wedge \varphi)$ and $\mu Z.(\varphi \vee ([\top]Z \wedge \langle\top\rangle\top))$, respectively. Absence of deadlock, i.e. the ability to always execute an action, is thus expressed as $\nu Z.([\top]Z \wedge \langle\top\rangle\top)$. A popular interpretation, due to Stirling and Bradfield (see e.g. [8]) is to think of a least fixpoint as *finite* looping through a set of states and to think of a greatest fixpoint as looping through a set of states. A list of standard patterns of properties can be found in e.g. [23].

The use of quantifiers inside modalities is illustrated by the following example. It shows how data-quantification in action formulae can be used for abstracting from the actual values for parameterised actions.

Example 7. Consider a process with at least the states s_0, s_1 and s_2, the labels $a(\top)$ and $a(\bot)$ and the μ-calculus formula φ. We write $s \models \varphi$ to denote that φ is satisfied in state s, and, likewise, we write $s \not\models \varphi$ to denote that φ is not satisfied in state s.

(1) The μ-calculus formula $\exists b{:}\mathbb{B}.\,[a(b)]\varphi$ holds in state s_0, since there is a b (viz. $b = \top$), such that whenever we execute $a(b)$, we end up in a state satisfying φ.
(2) The μ-calculus formula $[\exists b{:}\mathbb{B}.a(b)]\varphi$ does not hold in state s_0, since by executing $a(\bot)$ we end up in a state not satisfying φ. An alternative phrasing of the same property is $\forall b{:}\mathbb{B}.[a(b)]\varphi$.



One might think that the quantifiers inside modalities can all be replaced by quantifiers outside modalities. This, however, is only true for some combinations of modalities and quantifiers, as the following proposition testifies.

Proposition 8. *Let φ be a μ-calculus formula, such that d does not occur in φ, and let α be an action formula. Then, we have the following identities:*

(1) $\langle\exists d{:}D.\alpha\rangle\varphi \Leftrightarrow \exists d{:}D.\langle\alpha\rangle\varphi$, *and* $[\exists d{:}D.\alpha]\varphi \Leftrightarrow \forall d{:}D.[\alpha]\varphi$,
(2) $\exists d{:}D.[\alpha]\varphi \Rightarrow [\forall d{:}D.\alpha]\varphi$, *and* $\langle\forall d{:}D.\alpha\rangle\varphi \Rightarrow \forall d{:}D.\langle\alpha\rangle\varphi$.

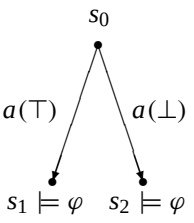
Note: here we use implication as an abbreviation for $\sqsubseteq$ on the interpretations of the μ-calculus formulae.

As a result of Proposition 8, we can conclude that, compared to the fragment of the μ-calculus that disallows quantifiers inside action formulae, the quantifiers inside action formulae indeed increase the expressive power of the calculus. However, it may not immediately be clear why the converse of the relations in item 2 do not hold. The following example sheds light on this.

Example 9. Assume again a process with at least the states s_0, s_1 and s_2, the labels $a(\top)$ and $a(\bot)$ and the state formula φ.

(1) The μ-calculus formula $\forall b{:}\mathbb{B}.\langle a(b)\rangle\varphi$ holds in state s_0. This is easily seen, as the universal quantifier ranges over a finite domain, and, thus, we can write this formula as $\langle a(\top)\rangle\varphi \wedge \langle a(\bot)\rangle\varphi$. However, the formula $\langle\forall b{:}\mathbb{B}.a(b)\rangle\varphi$ does not hold in state s_0: we can write this formula as $\langle\bot\rangle\varphi$, which actually holds in no state.

(2) Similarly, we can prove that the μ-calculus formula $[\forall b{:}\mathbb{B}.a(b)]\neg\varphi$ holds in state s_0. However, the formula $\exists b{:}\mathbb{B}.[a(b)]\neg\varphi$ does not hold in state s_0, since both transition $a(\top)$ and $a(\bot)$ lead to a state where φ holds, contradicting the requirement that φ should not hold.



Thus far, we have concentrated mainly on the regular μ-calculus constructs, and the use of quantifiers in this language. As our last example, we show how the parameterised fixpoints can be applied.

Example 10. Consider a system that can read natural numbers from a data stream using action r, see process X.

$$\textbf{\textit{proc}}\ X(n{:}\mathbb{N}) = \sum_{m:\mathbb{N}} r(m)\cdot X(m).$$

As this process can read from arbitrary streams of data, it should also be able to read a stream of *ascending natural numbers*. This property can be expressed by means of the following μ-calculus formula: $(\nu Z(i{:}\mathbb{N}).\exists j{:}\mathbb{N}\ \langle r(j)\rangle(i \leq j \wedge Z(j)))(0)$. Basically, since the μ-calculus is a state-based formalism, the parameter i in this formula records the last read value from the input stream, and uses this to compare against a newly read value via variable j. The formula is evaluated when using the value 0 as the initial value for i, i.e. the input is compared against 0, the smallest natural number. Note that the μ-calculus does not always need such variables to record information about histories: the property "a number never appears twice" is expressed by the μ-calculus formula $\nu Z.([\top]Z \wedge \forall j{:}\mathbb{N}\,[r(j)]\nu Z'.([\top]Z' \wedge \forall j'{:}\mathbb{N}\,[r(j')](j' \neq j)))$.

4. Equation systems

Following [21,12], the problem of model checking μ-calculus formulae over μCRL processes is transformed to the problem of solving *(first order) boolean equation systems*.

Whereas in [12], the authors define four deduction rules for verifying systems by hand, we follow Mader [21], and aim at automatic verification of systems. In this section, we review the theory for first order boolean equation systems, or *equation systems* for short. We discuss a number of lemmata that are subsequently used in our semi-decision procedure, given in Section 5.

Definition 11 (First Order Boolean Expression). A *first order boolean expression* is a formula φ in positive form, defined by the following grammar:

$$\varphi ::= b \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid X(e) \mid \forall d{:}D.\varphi \mid \exists d{:}D.\varphi$$

where b is an expression of domain $\mathbb{B}$, possibly containing data variables d from a set $\mathcal{D}$, e is a data expression (possibly containing data variables d of the set $\mathcal{D}$) of type D and X is a variable of a set $\mathcal{X}$ of (parameterised) propositional variables.

Note that first order boolean expressions are again open terms. The propositional variables $X \in \mathcal{X}$, occurring as free variables in first order boolean expressions are used in *equation systems*. First order boolean expressions are interpreted in the context of a propositional environment and a data environment.

Definition 12 (Semantics of First Order Boolean Expression). Let $\theta{:}\mathcal{X} \to (D \to \mathbb{B})$ be a propositional environment and $\eta{:}\mathcal{D} \to D$ be a data environment. The *interpretation* of a first order boolean expression φ in the context of environments θ and η, written as $[\![\varphi]\!]\theta\eta$ is either true or false, determined by the following induction:

$$\begin{aligned}
[\![b]\!]\theta\eta &= [\![b]\!]\eta \\
[\![X(e)]\!]\theta\eta &= \theta(X)([\![e]\!]\eta) \\
[\![\varphi_1 \wedge \varphi_2]\!]\theta\eta &= [\![\varphi_1]\!]\theta\eta \wedge [\![\varphi_2]\!]\theta\eta \\
[\![\varphi_1 \vee \varphi_2]\!]\theta\eta &= [\![\varphi_1]\!]\theta\eta \vee [\![\varphi_2]\!]\theta\eta \\
[\![\forall d{:}D.\varphi]\!]\theta\eta &= \begin{cases} \text{true, if for all } v{:}D \text{ it holds that } [\![\varphi]\!]\theta(\eta[v/d]) \\ \text{false, otherwise} \end{cases} \\
[\![\exists d{:}D.\varphi]\!]\theta\eta &= \begin{cases} \text{true, if there exists an } v{:}D \text{ such that } [\![\varphi]\!]\theta(\eta[v/d]) \\ \text{false, otherwise.} \end{cases}
\end{aligned}$$

We order first order boolean expressions by the inequality $\Rrightarrow$, defined as $\varphi \Rrightarrow \psi$ iff for all $\theta{:}\mathcal{X} \to (D \to \mathbb{B})$ and $\eta{:}D \to \mathbb{B}$, we have $[\![\varphi]\!]\theta\eta$ implies $[\![\psi]\!]\theta\eta$. Using this ordering, the set of all first order boolean expressions forms a complete lattice. We define an ordering $\leq$ on propositional environments as follows: $\theta_1 \leq \theta_2$ iff for all $X \in \mathcal{X}$, we have $\theta_1(X) \Rrightarrow \theta_2(X)$. The set of propositional environments, denoted $[\mathcal{X} \to (D \to \mathbb{B})]$, together with the ordering $\leq$ is also a complete lattice.

Definition 13 (Equation System). A *(first order boolean) equation system* $\mathcal{E}$ is a finite sequence of equations of the form $\sigma X(d{:}D) = \varphi$. Here, σ represents either the greatest or least fixpoints ν or μ, and φ is a first order boolean expression. We require that all data variables are bound exactly once and all bound propositional variables are unique. The empty equation system is denoted ϵ.

The equation system $\mathcal{E}'$ that is obtained by applying a propositional environment θ to an equation system $\mathcal{E}$ is the equation system in which every free propositional variable $X \in \mathcal{X}$ is assigned the value $\theta(X)$.

Definition 14 (Solution to an Equation System). Given a propositional environment θ, and an equation system $\mathcal{E}$. The solution $[\mathcal{E}]\theta$ to the equation system $\mathcal{E}$ is an environment that is defined as follows (see also e.g. Definition 3.3 in [21]), where σ is either the greatest fixpoint or the least fixpoint ν or μ.

$$\begin{array}{lcl} [\epsilon]\theta & = & \theta \\ [(\nu X(d{:}D) = \varphi)\mathcal{E}]\theta & = & [\mathcal{E}](\theta[\bigvee\{\psi{:}D \to \mathbb{B} \mid \psi \Rrightarrow \lambda d{:}D.[\![\varphi]\!]([\mathcal{E}]\theta[\psi/X])\}/X]) \\ [(\mu X(d{:}D) = \varphi)\mathcal{E}]\theta & = & [\mathcal{E}](\theta[\bigwedge\{\psi{:}D \to \mathbb{B} \mid \lambda d{:}D.[\![\varphi]\!]([\mathcal{E}]\theta[\psi/X]) \Rrightarrow \psi\}/X]). \end{array}$$

The operators $\bigwedge$ and $\bigvee$ denote the *greatest lower bound* and the *least upper bound* of the complete lattice of first order boolean expressions, respectively.

Lemma 15 (Monotonicity). *Let $\theta{:}\mathcal{X} \to (D \to \mathbb{B})$ be a propositional environment and $\eta{:}\mathcal{D} \to D$ a data environment. Define for an equation $\sigma X(d{:}D) = \varphi$, its associated operator $\Phi_{\theta\eta}{:}(D \to \mathbb{B}) \to (D \to \mathbb{B})$ as $\Phi_{\theta\eta} = \lambda F{:}D \to \mathbb{B}.\lambda v{:}D.[\![\varphi]\!](\theta[F/X])(\eta[v/d])$. The operator Φ is monotonic over the complete lattice of first order boolean expressions.*

Proof. Let $\theta{:}\mathcal{X} \to (D \to \mathbb{B})$ be a propositional environment and $\eta{:}\mathcal{D} \to D$ be a data environment. Let $\sigma X(d{:}D) = \varphi$ be an equation with associated functional $\Phi_{\theta\eta}$. Let ψ_1 and ψ_2 be arbitrary first order boolean expressions for which we have $\psi_1 \Rrightarrow \psi_2$. We proceed by induction on the structure of φ in the functional $\Phi_{\theta\eta}$.

- Suppose $\varphi \equiv b$. Then, $\Phi(\psi_1)$ equals $\lambda v{:}D.[\![b]\!](\theta[\psi_1/X])(\eta[v/d])$. By definition, this is equivalent to $\lambda v{:}D.[\![b]\!](\eta[v/d])$, and also to $\lambda v{:}D.[\![b]\!](\theta[\psi_2/X])(\eta[v/d])$. Therefore, we have $\Phi(\psi_1) \Rrightarrow \Phi(\psi_2)$.
- Suppose $\varphi \equiv Y(e)$. We distinguish between $Y \equiv X$ and $Y \not\equiv X$. For the latter case, we immediately obtain $\Phi(\psi_1) \Rrightarrow \Phi(\psi_2)$. Thus, we assume $Y \equiv X$. Then, $\Phi(\psi_1)$ can be written as $\lambda v{:}D.\psi_1([\![e]\!])(\eta[v/d])^{(*)}$, given that $Y \equiv X$ (if not, then we are done immediately). Since $\psi_1 \Rrightarrow \psi_2$, this is at most $\lambda v{:}D.\psi_2([\![e]\!])(\eta[v/d])$, which is equal to $\Phi(\psi_2)$. Thus, we have $\Phi(\psi_1) \Rrightarrow \Phi(\psi_2)$.
- Suppose $\varphi \equiv \varphi_1 \wedge \varphi_2$. Assume for first order boolean expressions φ_1 and φ_2, we already have $\Phi_1(\psi_1) \Rrightarrow \Phi_1(\psi_1)$ and $\Phi_2(\psi_1) \Rrightarrow \Phi_2(\psi_2)$. Then, $\Phi(\psi_1)$ is equal to the conjunction of the functionals $\lambda v{:}D.[\![\varphi_1]\!](\theta[\psi_1/X])(\eta[v/d])$ and $\lambda v{:}D.[\![\varphi_2]\!](\theta[\psi_1/X])(\eta[v/d])$. By induction, we know this is at most the conjunction of $\lambda v{:}D.[\![\varphi_1]\!](\theta[\psi_2/X])(\eta[v/d])$ and $\lambda v{:}D.[\![\varphi_2]\!](\theta[\psi_2/X])(\eta[v/d])$, which is equal to $\Phi(\psi_2)$. Thus, we know that $\Phi(\psi_1) \Rrightarrow \Phi(\psi_2)$. The case where we have $\varphi \equiv \varphi_1 \vee \varphi_2$ is similar.
- Suppose $\varphi \equiv \forall e{:}D.\varphi_e$. Assume for first order boolean expressions φ_e, we already have for all $e{:}D$, $\Phi_e(\psi_1) \leq \Phi_e(\psi_2)$. Then, $\Phi(\psi_1)$ is equal to the universal quantification over all $w{:}D$ in $\lambda v{:}D.[\![\varphi_e]\!](\theta[\psi_1/X])(\eta[v/d][w/e])$. By our induction hypothesis, this is at most the universal quantification over all $w{:}D$ in $\lambda v{:}D.[\![\varphi_e]\!](\theta[\psi_2/X])(\eta[v/d][w/e])$, which is equal to $\Phi(\psi_2)$. Thus, we have $\Phi(\psi_1) \Rrightarrow \Phi(\psi_2)$. The case where we have $\varphi \equiv \exists e{:}D.\varphi_e$ is similar.

Summarising, all cases lead to $\Phi(\psi_1) \Rightarrow \Phi(\psi_2)$, and thus, the functional Φ is a monotonic operator over first order boolean expressions. □

Lemma 16. *Equation systems are monotonic over the set of all propositional environments.*

Proof. Let $\theta_1, \theta_2 : \mathcal{X} \to (D \to \mathbb{B})$ be arbitrary propositional environments for which $\theta_1 \leq \theta_2$ holds. We use induction on the length of the equation system $\mathcal{E}$.

- Suppose $\mathcal{E} = \epsilon$. Then, by definition $[\epsilon]\theta_1 \leq [\epsilon]\theta_2$ holds.
- Suppose $\mathcal{E}$ is of the form $(\sigma X(d{:}D) = \varphi)\mathcal{E}$, and assume $[\mathcal{E}]\theta_1 \leq [\mathcal{E}]\theta_2$. Then $[(\sigma X(d{:}D) = \varphi)\mathcal{E}]\theta_1$ is, by definition equivalent to $[\mathcal{E}]\theta_1[\psi/X]$, where ψ abbreviates the solution to the equation for X, see Definition 14. Since we have $\theta_1 \leq \theta_2$, we also have $\theta_1[\psi/X] \leq \theta_2[\psi/X]$. Thus, we can apply our induction hypothesis to find that $[\mathcal{E}]\theta_1[\psi/X] \leq [\mathcal{E}]\theta_2[\psi/X]$. Then, we can rewrite $[\mathcal{E}]\theta_2[\psi/X]$ to $[(\sigma X(d{:}D) = \varphi)\mathcal{E}]\theta_2$.

Both cases lead to the required $[(\sigma X(d{:}D) = \varphi)\mathcal{E}]\theta_1 \leq [(\sigma X(d{:}D) = \varphi)\mathcal{E}]\theta_2$, and thus, equation systems are monotonic over the set of propositional environments. □

We next define a translation that converts any combination of μ-calculus formulae and μCRL LPEs to an equation system. In [12], it is shown that this translation has the property that it encodes the model checking problem in the problem of solving equation systems.

Definition 17. Let $\varphi = (\sigma Z(d_f{:}D).\Phi)(e)$ be a first order modal μ-calculus formula. Let $X(d_p{:}D)$ be an LPE (see Definition 2). The equation system $\mathcal{E}$ that corresponds to the expression φ for LPE X is given by $\mathbf{E}(\varphi)$. The translation function $\mathbf{E}$ is defined by structural induction in Table 1.

We illustrate the translation by means of a small example.

Example 18. We consider again the system of Example 10 that reads natural numbers from a data stream, see below. Note that this process is already in LPE form.

$$\textbf{\textit{proc}}\ X(n{:}\mathbb{N}) = \sum_{m:\mathbb{N}} r(m) \cdot X(m).$$

Suppose we are interested in whether we always eventually can read a 5 via action r. Formally, this takes on the following form

$$\nu Z.([\top]Z \wedge \mu Z'.(([\top]Z' \wedge \langle\top\rangle\top) \vee \langle r(5)\rangle\top)).$$

Remark here that this property expresses that a 5 *can* be read via action r. It must not be confused with the stronger property that value 5 *must* be read via action r.[1] Combining both LPE and the above modal formula, we obtain the following equation system:

$$(\nu \tilde{Z}(n{:}\mathbb{N}) = (\forall m{:}\mathbb{N}.\tilde{Z}(m)) \wedge \tilde{Z}'(n))$$
$$(\mu \tilde{Z}'(n{:}\mathbb{N}) = (\forall m{:}\mathbb{N}.\tilde{Z}'(m) \wedge \exists m{:}\mathbb{N}.\top) \vee \exists m{:}\mathbb{N}.m = 5).$$

[1] This stronger property would be expressed as $\nu Z.[\neg r(5)]Z \wedge \langle\top\rangle\top$. Whenever we refer to an "eventually" property in this paper, we refer to the weaker version of the "eventually" property.

Table 1
Translation of first order μ-calculus formula φ and LPE $X(d_p{:}D)$ to an equation system $\mathbf{E}(\varphi)$

$\mathbf{E}(b)$	$\stackrel{\text{def}}{=}$	ϵ
$\mathbf{E}(Z(d_f))$	$\stackrel{\text{def}}{=}$	ϵ
$\mathbf{E}(\Phi_1 \wedge \Phi_2)$	$\stackrel{\text{def}}{=}$	$\mathbf{E}(\Phi_1)\mathbf{E}(\Phi_2)$
$\mathbf{E}(\Phi_1 \vee \Phi_2)$	$\stackrel{\text{def}}{=}$	$\mathbf{E}(\Phi_1)\mathbf{E}(\Phi_2)$
$\mathbf{E}([\alpha]\Phi)$	$\stackrel{\text{def}}{=}$	$\mathbf{E}(\Phi)$
$\mathbf{E}(\langle\alpha\rangle\Phi)$	$\stackrel{\text{def}}{=}$	$\mathbf{E}(\Phi)$
$\mathbf{E}(\forall d{:}D.\Phi)$	$\stackrel{\text{def}}{=}$	$\mathbf{E}(\Phi)$
$\mathbf{E}(\exists d{:}D.\Phi)$	$\stackrel{\text{def}}{=}$	$\mathbf{E}(\Phi)$
$\mathbf{E}((\sigma Z(d_f{:}D_f).\Phi)(e))$	$\stackrel{\text{def}}{=}$	$(\sigma \tilde{Z}(d_f{:}D_f, d_p{:}D_p, \mathbf{Par}_Z(\varphi)) = \mathbf{RHS}(\Phi))\ \mathbf{E}(\Phi)$
$\mathbf{RHS}(b)$	$\stackrel{\text{def}}{=}$	b
$\mathbf{RHS}(Z(e))$	$\stackrel{\text{def}}{=}$	$\tilde{Z}(e, d_p, \mathbf{Par}_Z(\varphi))$
$\mathbf{RHS}(\Phi_1 \wedge \Phi_2)$	$\stackrel{\text{def}}{=}$	$\mathbf{RHS}(\Phi_1) \wedge \mathbf{RHS}(\Phi_2)$
$\mathbf{RHS}(\Phi_1 \vee \Phi_2)$	$\stackrel{\text{def}}{=}$	$\mathbf{RHS}(\Phi_1) \vee \mathbf{RHS}(\Phi_2)$
$\mathbf{RHS}([\alpha]\Phi)$	$\stackrel{\text{def}}{=}$	$\bigwedge_{i:I} \forall_{e_i:D_i}(a_i(f_i(d_p, e_i)) \models \alpha \wedge c_i(d_p, e_i)) \Rightarrow \mathbf{RHS}(\Phi)[g_i(d_p, e_i)/d_p]$
$\mathbf{RHS}(\langle\alpha\rangle\Phi)$	$\stackrel{\text{def}}{=}$	$\bigvee_{i:I} \exists_{e_i:D_i}(a_i(f_i(d_p, e_i)) \models \alpha \wedge c_i(d_p, e_i) \wedge \mathbf{RHS}(\Phi)[g_i(d_p, e_i)/d_p])$
$\mathbf{RHS}(\forall d{:}D.\Phi)$	$\stackrel{\text{def}}{=}$	$\forall d{:}D.\mathbf{RHS}(\Phi)$
$\mathbf{RHS}(\exists d{:}D.\Phi)$	$\stackrel{\text{def}}{=}$	$\exists d{:}D.\mathbf{RHS}(\Phi)$
$\mathbf{RHS}((\sigma Z(d_f{:}D_f).\Phi)(e))$	$\stackrel{\text{def}}{=}$	$\tilde{Z}(e, d_p, \mathbf{Par}_Z(\varphi))$

Note that $\tilde{Z}$ is a fresh propositional variable, associated with the propositional variable Z. Function $\mathbf{E}$ determines the number and order of equations for $\mathbf{E}(\varphi)$, whereas function **RHS** breaks down φ to obtain first order boolean expressions that form the right-hand side of each equation in $\mathbf{E}(\varphi)$. The satisfaction relation $\models$ and the function **Par** are listed in Table 2. The function $\mathbf{Par}_X(\varphi)$ yields a list of parameters with types that must be bound by the parameterised propositional variable X. Here, we have abused the notation $\mathbf{Par}_X(\varphi)$ to also denote the list of parameters without typing information. Note that $\mathbf{Par}_X(\varphi)$ is always calculated for the entire formula φ, and not for subformulae.

Note that, even though the modal formula itself does not carry any parameters, the parameter n stems from the LPE X. Obviously, the resulting equation system can be further simplified using rules of calculation, see e.g. [12,17]. This, however, is not the objective of this paper.

The following result, due to Groote and Mateescu [12, Proposition 1], confirms the relation between the model checking problem and the problem of solving equation systems.

Theorem 19 (Groote and Mateescu [12]). *Translating an LPE X and μ-calculus formula φ using the function* **E** *of Definition 17 has the property that the solution to the resulting equation system is true iff process X satisfies property φ.*

Table 2
Auxiliary functions used in the translation of Table 1

$$
\begin{array}{lcl}
a(d) \models a'(d') & \stackrel{\text{def}}{=} & a = a' \wedge d = d' \\
a(d) \models \top & \stackrel{\text{def}}{=} & \text{true} \\
a(d) \models \neg\alpha & \stackrel{\text{def}}{=} & \neg(a(d) \models \alpha) \\
a(d) \models \alpha_1 \wedge \alpha_2 & \stackrel{\text{def}}{=} & (a(d) \models \alpha_1) \wedge (a(d) \models \alpha_2) \\
a(d) \models \alpha_1 \vee \alpha_2 & \stackrel{\text{def}}{=} & (a(d) \models \alpha_1) \vee (a(d) \models \alpha_2) \\
a(d) \models \exists d'{:}D.\alpha & \stackrel{\text{def}}{=} & \exists d'{:}D.(a(d) \models \alpha) \\
a(d) \models \forall d'{:}D.\alpha & \stackrel{\text{def}}{=} & \forall d'{:}D.(a(d) \models \alpha)
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{Par}_X(b) & \stackrel{\text{def}}{=} & [] \\
\mathbf{Par}_X(Z(d_f)) & \stackrel{\text{def}}{=} & [] \quad \text{for all } Z{\in}\mathcal{P} \\
\mathbf{Par}_X(\Phi_1 \wedge \Phi_2) & \stackrel{\text{def}}{=} & \mathbf{Par}_X(\Phi_1) +\!\!+ \mathbf{Par}_X(\Phi_2) \\
\mathbf{Par}_X(\Phi_1 \vee \Phi_2) & \stackrel{\text{def}}{=} & \mathbf{Par}_X(\Phi_1) +\!\!+ \mathbf{Par}_X(\Phi_2) \\
\mathbf{Par}_X([\alpha]\Phi) & \stackrel{\text{def}}{=} & \mathbf{Par}_X(\Phi) \\
\mathbf{Par}_X(\langle\alpha\rangle\Phi) & \stackrel{\text{def}}{=} & \mathbf{Par}_X(\Phi) \\
\mathbf{Par}_X(\forall d{:}D.\Phi) & \stackrel{\text{def}}{=} & [d{:}D] +\!\!+ \mathbf{Par}_X(\Phi) \\
\mathbf{Par}_X(\exists d{:}D.\Phi) & \stackrel{\text{def}}{=} & [d{:}D] +\!\!+ \mathbf{Par}_X(\Phi) \\
\mathbf{Par}_X((\sigma Z(d_f{:}D_f).\Phi)(e)) & \stackrel{\text{def}}{=} & [d_f{:}D_f] +\!\!+ \mathbf{Par}_X(\Phi) \quad \text{for all } Z \neq X \\
\mathbf{Par}_X((\sigma X(d_f{:}D_f).\Phi)(e)) & \stackrel{\text{def}}{=} & []
\end{array}
$$

Here, $+\!\!+$ denotes list concatenation. The satisfaction relation $\models$ checks whether a symbolic action $a(d)$ is part of an action formula α. The function $\mathbf{Par}_X(\varphi)$ yields a list of parameters together with their types that have to be bound by the equation for X.

Whereas Groote and Mateescu [12] use this result to solve the model checking problem by means of manual verification using four deduction rules, we use this result as the basis for a semi-decision procedure. First, we establish three results that are at the basis of the soundness of this procedure.

Proposition 20. *Let $\mathcal{E}, \mathcal{E}_1$ and $\mathcal{E}_2$ be equation systems. If for all environments θ, $[\mathcal{E}_1]\theta = [\mathcal{E}_2]\theta$ then for all environments η, $[\mathcal{E}\mathcal{E}_1]\eta = [\mathcal{E}\mathcal{E}_2]\eta$.*

Proof. By induction on the length of $\mathcal{E}$. □

The following lemma states that for an arbitrary equation system, we may replace an occurrence of an equation variable with its first order boolean expression in all equations prior to its defining equation.

Lemma 21. *Let $\mathcal{E}_1, \mathcal{E}_2$ and $\mathcal{E}_3$ be equation systems and let $\sigma_1 X_1(d_1{:}D_1) = \varphi$ and $\sigma_2 X_2(d_2{:}D_2) = \psi$ be equations. Then, we can substitute ψ in all preceding equations in which X_2 occurs on the right-hand side, i.e.*

$$\begin{aligned}&[\mathcal{E}_1(\sigma_1 X_1(d_1{:}D_1) = \varphi)\mathcal{E}_2(\sigma_2 X_2(d_2{:}D_2) = \psi)\mathcal{E}_3]\theta\\ =\\ &[\mathcal{E}_1(\sigma_1 X_1(d_1{:}D_1) = \varphi[\psi/X_2])\mathcal{E}_2(\sigma_2 X_2(d_2{:}D_2) = \psi)\mathcal{E}_3]\theta.\end{aligned}$$

Proof. Using Proposition 20, it is easy to see that it suffices to prove that

$$\begin{aligned}&[(\sigma_1 X_1(d_1{:}D_1) = \varphi_1)\mathcal{E}_2(\sigma_2 X_2(d_2{:}D_2) = \varphi_2)\mathcal{E}_3]\theta\\ =\\ &[(\sigma_1 X_1(d_1{:}D_1) = \varphi_1[\varphi_2/X_2])\mathcal{E}_2(\sigma_2 X_2(d_2{:}D_2) = \varphi_2)\mathcal{E}_3]\theta.\end{aligned}$$

This follows directly from the following observation:

$$\varphi_1([\mathcal{E}_2(\sigma_2 X_2(d_2{:}D_2) = \psi)\mathcal{E}_3]\theta) = \varphi_1[\psi/X_2]([\mathcal{E}_2(\sigma_2 X_2(d_2{:}D_2) = \psi)\mathcal{E}_3]\theta)$$

which can easily be shown to hold using induction on the length of $\mathcal{E}_2$. □

A single equation for which we know its solution, can be removed from an equation system by updating the propositional environment. This means that by successively solving all single equations, the solution to the entire equation system follows.

Lemma 22. *Let $\mathcal{E}, \mathcal{E}'$ be equation systems and let $\sigma X(d{:}D) = \varphi$ be an equation, where X does not occur freely in φ and let θ be an arbitrary propositional environment. Then*

$$[\mathcal{E}(\sigma X(d{:}D) = \varphi)\mathcal{E}']\theta = [\mathcal{E}\mathcal{E}']\theta[\varphi/X].$$

Proof. Note that since X does not occur in φ, the solution to the fixpoint equation $\sigma X(d{:}D) = \varphi$ is itself again φ. We proceed by induction on the size of the equation system $\mathcal{E}$.

- Suppose $\mathcal{E} = \epsilon$. By definition, $[(\sigma X(d{:}D) = \psi)\mathcal{E}']\theta$ is equal to $[\mathcal{E}'](\theta[\psi/X])$,
- Suppose $\mathcal{E}$ is of the form $(\sigma' X'(d'{:}D') = \varphi')\mathcal{E}_0$, and assume $[\mathcal{E}_0(\sigma X(d{:}D) = \varphi)\mathcal{E}']\theta = [\mathcal{E}_0\mathcal{E}']\theta[\varphi/X]$ for all environments θ. We denote the solution to X' by $\sigma' X'(d'{:}D').\varphi'$. Then, $[(\sigma' X'(d'{:}D') = \varphi')\mathcal{E}_0(\sigma X(d{:}D) = \varphi)\mathcal{E}']\theta$ is equal to $[\mathcal{E}_0(\sigma X(d{:}D) = \varphi)\mathcal{E}'](\theta[\sigma' X'(d'{:}D').\varphi'/X'])$. By structural induction, this is equivalent to $[\mathcal{E}_0\mathcal{E}']((\theta[\sigma' X'(d'{:}D').\varphi'/X'])[\varphi/X])$. By definition, this is equal to $[\mathcal{E}\mathcal{E}']\theta[\psi/X]$.

  Summarising, both cases lead to the desired equality. □

5. Decision procedure

Mader [21, Chapter 6] describes an algorithm for solving boolean equation systems. The method she uses resembles the well-known Gauß elimination algorithm for solving linear equation systems, and is therefore also referred to as Gauß elimination. The semi-decision procedure we use (see Fig. 1) is an extension of the Gauß elimination algorithm

of [21]. The essential difference is in the addition of an extra loop for calculating a fixpoint in the approximation for each equation.

We briefly explain the procedure. The reduction of an equation system proceeds in two separate steps. First, a stabilisation step is issued (see line 3), in which an equation $\sigma_i X_i(d{:}D) = \varphi_i$ is reduced to a stable equation $\sigma_i X_i(d{:}D) = \varphi_i'$, where φ_i' is an expression containing no occurrences of X_i. Second, we substitute each occurrence of X_i by φ_i' in the rest of the equations of the equation system (line 4). Since there are no more occurrences of X_i in the right-hand side of the equations, it suffices to reduce a smaller equation system. Lines 1 and 5 are needed for performing the substitutions starting with the last equation of the equation system and working down to the first equation. In line 2 the basis for the approximation step (in line 3) is made by assigning $\top$ to a greatest fixpoint equation and $\bot$ to a least fixpoint equation. The semi-decision procedure terminates iff the stabilisation step terminates for each equation.

Input: $(\sigma_1 X_1(d_1{:}D_1) = \varphi_1) \ldots (\sigma_n X_n(d_n{:}D_n) = \varphi_n)$.

1. **for** $i = n$ **downto** 1 **do**
2. $\quad j := 0$; $\psi_0 :=$ (**if** $\sigma_i = \nu$ **then** $\top$ **else** $\bot$);
3. $\quad$ **repeat** $\psi_{j+1} := \varphi_i[X_i := \psi_j]$; $j := j + 1$ **until** $(\psi_j \equiv \psi_{j-1})$
4. $\quad$ **for** $k = 1$ **to** i **do** $\varphi_k := \varphi_k[X_i := \psi_j]$ **od** ;
5. **od**

Fig. 1. Semi-decision procedure for computing the solution of an equation system.

Theorem 23 (Soundness). *On termination of the procedure in Fig. 1, the solution of the given equation system has been computed.*

Proof. The technique to solve a single equation is based on well-established transfinite approximation techniques [19]. Termination of this approximation means we have computed a solution to a single equation. This solution can then be substituted in the remainder of the equation system, as a result of Lemmas 21 and 22. Termination of the procedure, therefore, means we have correctly computed the solution to all equations in the equation system. □

Note that as it is undecidable whether an equation system has a solution, the possible non-termination of our procedure is unavoidable. Note that the decidability depends largely on the data types that are used and not so much on the class of (first order) modal μ calculus formulae. For instance, the alternation-free fragment of the modal μ-calculus (see e.g. [22]) still allows for coding the halting problem. Below, we illustrate how the model checking problem of various small data-dependent systems is solved using the translation of Definition 17 and the procedure we defined in this section.

Example 24. Consider a counter that counts up to nine, starting from zero, and at nine cycles back to zero. Each time the counter increases, an *inc* event is issued. Upon reaching nine, the counter issues a *reset* event, signalling the counter has been reset to zero. A process algebraic description (in LPE form) of such a process is provided below.

$$\begin{aligned} \textbf{\textit{proc}}\ C(n{:}\mathbb{N}) \;&=\; [n \geq 9]{:}\!\rightarrow \mathit{reset} \cdot C(0) \\ &+\; [n < 9]{:}\!\rightarrow \mathit{inc} \cdot C(n+1). \end{aligned}$$

Our goal is to verify whether it is possible to eventually execute a *reset* action (the process obviously does not deadlock, so, in all states, the formula $\langle\top\rangle\top$ is immediately satisfied. Thus, we can leave this part out of the standard construction for "eventually reset"). Expressed formally, we obtain the formula $\mu Z.[\top]Z \vee \langle \mathit{reset}\rangle\top$. This basically expresses that on all paths, eventually a *reset* action is executed. The equation system for this expression is (after removing some redundancies) $\mu\tilde{Z}(n{:}\mathbb{N}) = (n \geq 9 \vee \tilde{Z}(n+1))$.

Following the procedure, we first compute the first and second approximations ψ_0 and ψ_1, being $\bot$ and $n \geq 9$, respectively. Then, we iterate until we end up with a formula $\psi_{10} = 0 \leq n$, which is equivalent to ψ_{11}. Since this is a stable solution of the equation, we can assess the truth of the equation system by substituting ψ_{10} for $\tilde{Z}$ in our equation, thereby obtaining $\mu\tilde{Z}(n{:}\mathbb{N}) = \top$. Verifying the validity of this property in an arbitrary initial state n' of the system immediately yields true.

Example 25. As an example of a system with an infinite state-space, we consider a process that counts from zero to infinity, and reports its current state via an action *current*. A process algebraic description in LPE form is provided below.

$$\textbf{\textit{proc}}\ C(n{:}\mathbb{N}) = \mathit{current}(n) \cdot C(n+1).$$

Given the simplicity of this process, it is unfortunate to find that with most current technologies, proving absence of deadlock for process C cannot be done automatically. Using our procedure, this boils down to verifying $\nu X.(\langle\top\rangle\top \wedge [\top]X)$ on the process C. Following the translation, we derive the associated equation system $\nu\tilde{Z}(n{:}\mathbb{N}) = (\tilde{Z}(n+1) \wedge \top)$. The first approximation, being $\top$ is immediately stable (i.e. the second approximation is also $\top$). Hence, the solution to this equation is $\top$.

Last but not least, we present an example of a system for which the model checking procedure we defined in this section does not terminate.

Example 26. Consider a process C representing a counter that counts down from a randomly chosen natural number to zero and then randomly selects a new natural number.

$$\begin{aligned} \textbf{\textit{proc}}\ C(n{:}\mathbb{N}) \;&=\; \sum_{m:\mathbb{N}} [n = 0]{:}\!\rightarrow \mathit{reset} \cdot C(m) \\ &+\; [n > 0]{:}\!\rightarrow \mathit{dec} \cdot C(n-1). \end{aligned}$$

We verify whether it is possible to eventually execute a *reset* action. Since the process obviously is deadlock free, this can be expressed using the standard construction as $\mu Z.[\top]Z \vee \langle \mathit{reset}\rangle\top$. The equation system associated with this expression is $\mu\tilde{Z}(n{:}\mathbb{N}) = ((n > 0 \Rightarrow \tilde{Z}(n-1)) \wedge \forall m{:}\mathbb{N}(n = 0 \Rightarrow \tilde{Z}(m)) \vee n = 0)$.

The procedure prescribes computing a stable solution for this equation. However, this computation does not terminate, as we end up with approximations ψ_k, where $\psi_k = n \leq k$.

This means, we cannot find a ψ_j, such that $\psi_j = \psi_{j+1}$, and therefore, the procedure does not terminate. However, it is straightforward to see that the minimal solution for this equation is necessarily $\mu\tilde{Z}(n{:}\mathbb{N}) = \top$. This solution can be found using the results described in [17].

6. Verification of data-dependent systems in practice

Based on our algorithm, described in the previous section, we have implemented a prototype of a tool.[2] In this section, we briefly sketch this implementation, without going into detail.

6.1. Implementation

The prototype implementation of our algorithm employs *Equational Binary Decision Diagrams* (EQ-BDDs) [16] for representing first order boolean expressions. These EQ-BDDs extend on standard BDDs [9] by explicitly allowing equality on nodes. We first define the grammar for EQ-BDDs.

Definition 27 (Grammar for EQ-BDDs). We assume a set P of propositions and a set V of variables. The formulae we consider are given according to the following grammar.

$$\Phi ::= \mathbf{0} \mid \mathbf{1} \mid \mathrm{ITE}(V = V, \Phi, \Phi) \mid \mathrm{ITE}(P, \Phi, \Phi).$$

The constants **0** and **1** represent *false* and *true*. Expressions of the form $\mathrm{ITE}(\varphi, \psi, \xi)$ must be read as *if–then–else* constructs, i.e. $(\varphi \wedge \psi) \vee (\neg\varphi \wedge \xi)$, or, alternatively, $(\varphi \Rightarrow \psi) \wedge (\neg\varphi \Rightarrow \xi)$. For data variables d and e, and φ of the form $d = e$, the extension to EQ-BDDs is used, i.e. we explicitly use $\mathrm{ITE}(d = e, \psi, \xi)$ in such cases. Using the standard BDD and EQ-BDD encodings [9,16], we can then represent all quantifier-free first order boolean expressions. The representation of expressions that contain quantifiers over finite domains is done in a straightforward manner, i.e. we construct explicit encodings for each distinct element in the domain. Expressions containing quantifiers over infinite domains are in general problematic. The following theorem, identifies a number of cases in which we can deal with these.

Theorem 28. *Suppose data variable d does not occur in expression ψ. Then, quantification over data-types can be removed in the following cases:*

- $\exists d{:}D.ITE(d = e, \varphi, \psi) = \varphi[e/d] \vee \psi$ *provided D contains at least two elements.*
- $\forall d{:}D.ITE(d = e, \varphi, \psi) = \varphi[e/d] \wedge \psi$ *provided D contains at least two elements.*
- $\exists d{:}D.ITE(d = e_1, \varphi_1, ITE(d = e_2, \varphi_2, \ldots, ITE(d = e_n, \varphi_n, \psi)\ldots)) = \bigvee_{1\le i\le n}((\bigwedge_{1\le j<i} e_j \neq e_i) \wedge \varphi_i[e_i/d]) \vee \psi$ *provided D contains at least one element not in $\{e_i \mid 1 \le i \le n\}$.*

[2] This prototype implementation is freely available as part of the μCRL tool-suite [4], see the subdirectory *checker*.

- $\forall d{:}D.ITE(d = e_1, \varphi_1, ITE(d = e_2, \varphi_2, \ldots, ITE(d = e_n, \varphi_n, \psi)\ldots)) = \bigwedge_{1\leq i\leq n}((\bigvee_{1\leq j<i} e_j = e_i) \vee \varphi_i[e_i/d]) \wedge \psi$ *provided* D *contains at least one element not in* $\{e_i | 1 \leq i \leq n\}$.

Proof. The identities follow directly from the observations that

- $\exists d{:}D.\mathrm{ITE}(d = e, \varphi, \psi) = \varphi[e/d] \vee \exists d{:}D(d \neq e \wedge \psi)$.
- $\forall d{:}D.\mathrm{ITE}(d = e, \varphi, \psi) = \varphi[e/d] \wedge \forall d{:}D(d = e \vee \psi)$.

Obviously, Theorem 28 applies to a restricted class of first order boolean expressions. In practice, we find that the above theorem adds considerably to the verification power of the prototype implementation. This is illustrated in the next section.

7. Example verifications

We have successfully used the prototype on several applications, including many communications protocols, such as the IEEE-1394 firewire, sliding window protocols, the bounded retransmission protocol, etc. We first illustrate the performance of the tool on a small example,[3] viz. a one-place buffer that holds elements of an arbitrary (possibly infinite) domain M, equipped with equality (see Table 3). Let P be an arbitrary, total predicate on M.

Table 3
A simple one-place buffer

proc $Buffer = \sum_{m:M} read(m) \cdot send(m) \cdot Buffer$

We verify two properties, illustrating our techniques can deal with abstract notions such as arbitrary domains and predicates on such domains.

(1) If the input is constant, then the output is also constant, i.e.
$$\forall m{:}M.(\nu Z(i{:}M).\forall n{:}M.([read(n)](n = i \Rightarrow Z(n)) \wedge [send(n)](n = i \wedge Z(n))))(m).$$
(2) If the input satisfies predicate P, then the output also satisfies predicate P, i.e.
$$\forall k{:}M.(\nu Z(i, j{:}M).\forall n{:}M.([read(n)](P(n) \Rightarrow Z(n, j)) \wedge [send(n)](P(n) \wedge Z(i, n))))(k, k).$$

Using our prototype, the above properties are verified in less than 1 s, and both are proved to be satisfied.

We next discuss two larger examples. We first report on our findings for Lamport's Bakery Protocol [26]. A μCRL specification of the bakery protocol is given in Table 4.

The bakery protocol we consider is restricted to two processes. Each process, waiting to enter its critical section, can choose a number, larger than any other number already chosen.

[3] All results listed in this section were obtained using a 1.47 GHz AMD Athlon XP1700+ machine with 256 MB main memory, running Linux kernel 2.4.

Table 4
Lamport's bakery protocol

$$\textbf{comm}\ get, send = c$$
$$\textbf{init}\ \partial_{\{get,\ send\}}(P(\top)\|P(\bot))$$

$$\textbf{proc}\ P(b{:}\mathbb{B}) = request(b)\cdot P_0(b,0) + send(b,0)\cdot P(b)$$
$$\textbf{proc}\ P_0(b{:}\mathbb{B}, n{:}\mathbb{N}) = \textstyle\sum_{m:\mathbb{N}}\ get(\neg b, m)\cdot P_1(b, m+1) + send(b,n)\cdot P_0(b,n)$$
$$\textbf{proc}\ P_1(b{:}\mathbb{B}, n{:}\mathbb{N}) = send(b,n)\cdot P_1(b,n) + \textstyle\sum_{m:\mathbb{N}}\ get(\neg b, m)\cdot (C_1(b,n) \lhd n < m \vee m = 0 \rhd P_1(b,n))$$
$$\textbf{proc}\ C_1(b{:}\mathbb{B}, n{:}\mathbb{N}) = enter(b)\cdot C_2(b,n) + send(b,n)\cdot C_1(b,n)$$
$$\textbf{proc}\ C_2(b{:}\mathbb{B}, n{:}\mathbb{N}) = leave(b)\cdot P(b) + send(b,n)\cdot C_2(b,n)$$

Then, the process with the lower number is allowed to enter the critical section before the process with the larger number. Due to the unbounded growth of the numbers that can be chosen, the protocol has an infinite state-space. However, our techniques are immediately applicable. Below, we list a number of key properties we verify for the bakery protocol.

(1) No deadlocks, i.e.
$\nu Z.([\top]Z \wedge \langle\top\rangle\top)$,
(2) Two processes can never be in the critical section at the same time, i.e.
$\nu Z.([T]Z \wedge \forall b{:}\mathbb{B}.([enter(b)]\nu Z'.([enter(\neg b)]\bot \wedge [\neg leave(b)]Z')))$,
(3) All processes requesting a number always possibly enter the critical section, i.e.
$\nu Z.([\top]Z \wedge \forall b{:}\mathbb{B}.([request(b)]\mu Z'.(\langle\top\rangle Z' \vee \langle enter(b)\rangle\top)))$,
(4) All processes requesting a number always eventually enter the critical section, i.e.
$\nu Z.([\top]Z \wedge \forall b{:}\mathbb{B}.([request(b)]\mu Z'.(([\top]Z' \wedge \langle\top\rangle\top) \vee \langle enter(b)\rangle\top)))$
(5) It is always possible to get a number, i.e.
$\nu Z.([\top]Z \wedge \forall b{:}\mathbb{B}.\mu Z'.(([\top]Z' \wedge \langle\top\rangle\top) \vee \exists n{:}\mathbb{N}.\langle c(b,n)\rangle\top))$
(6) It is attainable for a process to always get a number that is at least the number that is currently circulating, i.e.
$(\nu Z(i{:}\mathbb{N}).(\exists b{:}\mathbb{B}.\exists n{:}\mathbb{N}.\langle c(b,n)\rangle(n \geq i \wedge Z(n))) \vee \langle\forall b'{:}\mathbb{B}.\forall n'{:}\mathbb{N}.\neg c(b',n')\rangle Z(i))(0)$.

All properties but the fourth are satisfied. Using our prototype we were able to produce the results for the first and third property in less than a second, the second property in 2 s and the fifth and sixth property in 3 and 19 s, respectively. The fourth property was proved not to hold by our prototype in 2 s.

Lastly, we consider the *Alternating Bit Protocol* (ABP) defined in Table 5 (cf. [3]). This is a basic communications protocol utilising two unreliable channels. A sender sends a message, tagged with a bit, via an unreliable channel. It repeatedly resends this message (including the bit), until it receives an acknowledgement (with the right bit) from the receiver, via the other channel. It then starts the entire procedure again with a new message, and inverts the bit it sends along with the message.

The ABP is a famous communications protocol, and is often used to illustrate that a formalism or technique is capable of dealing with real systems of small to medium size.

When applying well-established, fully automatic techniques, the data that is transmitted in this (and other) communications protocols, has to be fixed. Here, we show that, with the use of our prototype, no alterations to the ABP are necessary, and the messages we transmit are indeed arbitrarily chosen from an infinite set of messages.

Table 5
Alternating bit protocol

$$
\begin{array}{l}
\textbf{\textit{comm}}\ r2, s2 = c2; r3, s3 = c3; r5, s5 = c5; r6, s6 = c6\\
\textbf{\textit{init}}\ \partial_{\{r2,r3,r5,r6,s2,s3,s5,s6\}}(S \| K \| L \| R)\\
\\
\textbf{\textit{proc}}\ S = S(0) \cdot S(1) \cdot S\\
\textbf{\textit{proc}}\ S(n{:}bit) = \sum_{d:D}\ r1(d) \cdot S(d,n)\\
\textbf{\textit{proc}}\ S(d{:}D, n{:}bit) = s2(d,n) \cdot ((r6(\overline{n}) + r6(e)) \cdot S(d,n) + r6(n))\\
\textbf{\textit{proc}}\ R = R(1) \cdot R(0) \cdot R\\
\textbf{\textit{proc}}\ R(n{:}bit) = (r3(e) + \sum_{d:D}\ r3(d,n)) \cdot s5(n) \cdot R(n)\\
\qquad\qquad + \sum_{d:D}\ r3(d,\overline{n}) \cdot s4(d) \cdot s5(\overline{n})\\
\textbf{\textit{proc}}\ K = \sum_{d:D}\ \sum_{n:bit}\ r2(d,n) \cdot (i \cdot s3(d,n) + i \cdot s3(e)) \cdot K\\
\textbf{\textit{proc}}\ L = \sum_{n:bit}\ r5(n) \cdot (i \cdot s6(n) + i \cdot s6(e)) \cdot L
\end{array}
$$

Communications protocols usually have an external behaviour, similar to the behaviour of a buffer, i.e. messages sent at one end are eventually received at the other end. The ABP is no exception to this rule. The properties we verified for ABP are listed below.

(1) No deadlock can occur, i.e.
$\nu Z.([\top]Z \wedge \langle\top\rangle\top)$,
(2) A message that is received by the sender (via $r1$) always possibly will be sent by the receiver (via $s4$), i.e.
$\nu Z.([\top]Z \wedge \forall d{:}D.[r1(d)]\mu Z'.(\langle\top\rangle Z' \vee \langle s4(d)\rangle\top))$,
(3) A message that is received by the sender (via $r1$) always eventually will be sent by the receiver (via $s4$), i.e.
$\nu Z.([\top]Z \wedge \forall d{:}D.[r1(d)]\mu Z'.(([\top]Z' \wedge \langle\top\rangle\top) \vee \langle s4(d)\rangle\top))$,
(4) Every message that is received by the sender (via $r1$), can as long as it has not been delivered (via $s4$) eventually be delivered, i.e.
$\nu Z.([\top]Z \wedge \forall d{:}D.[r1(d)]\nu Z'.([\neg s4(d)]Z' \wedge \mu Z''.(\langle\top\rangle Z'' \vee \langle s4(d)\rangle\top)))$
(5) If the choice between losing a message and sending a message is resolved fairly, then the message is always eventually delivered (via $s4$), i.e.
$\nu Z.([\top]Z \wedge \forall d{:}D.[r1(d)]\nu Z'.\mu Z''.([c3(e)]Z' \wedge [\neg(c3(e) \vee s4(d))]Z'') \wedge \langle\top\rangle\top)$
(6) The protocol does not create messages, i.e.
$\nu Z.\forall d{:}D.([\neg r1(d)]X \wedge [s4(d)]\bot)$.
(7) The protocol does not duplicate messages, i.e.
$\nu Z.([\top]Z \wedge \forall d{:}D.[r1(d)]\nu Z'.([\neg(r1(d) \wedge s4(d))]Z' \wedge [s4(d)]\nu Z''.([\neg r1(d)]Z'' \wedge [s4(d)]\bot)))$.

The first two properties are shown to hold in 1 s and 15 s, respectively. The third property is shown not to hold, simply because the channel can infinitely often lose or scramble a message. Our prototype obtained this result in 4 s.

The fourth and fifth property are typical fairness properties. Our prototype produced the result that these properties are satisfied in 16 s and 14 s, respectively.

The sixth and seventh properties are "no-miracle" properties. Using our prototype, we were able to show that the ABP satisfies these properties in 1 s and 5 s, respectively.

8. Closing remarks

8.1. Related work

In a setting without data, the use of boolean equation systems for the verification of modal μ-calculus formulae on finite and infinite state systems was studied by Mader [21]. As observed by Mader, the use of boolean equation systems is closely related to the tableau methods of Bradfield and Stirling [7], but avoids certain redundancy of tableaux. It is therefore likely that in the case with data our approach performs better than tableau methods if these were extended to deal with data.

Closely related to our work is the tool EVALUATOR 3.0 [23], which is an on-the-fly model checker for the regular *alternation-free μ-calculus*, see also [22]. The machinery of this tool is based on boolean equation systems. The alternation-free μ-calculus is a fragment of the (first order) modal μ-calculus with alternation depth 1. Although the alternation-free μ-calculus allows for the specification of temporal logic properties involving data, the current version of the tool does not support the data-based version of this language. It is well imaginable that this tool can be extended with our techniques.

A different approach altogether is undertaken by e.g. Bryant et al. [10]. Their *Counter arithmetic with Lambda expressions and Uninterpreted function* (CLU) can be used to model both data and control, and is shown to be decidable. For this, CLU sacrifices expressiveness, as it is restricted to the quantifier-free fragment of first order logic. Moreover, their tool (UCLID) is restricted to dealing with safety properties only. We allow for safety, liveness and fairness properties to be verified automatically. Nevertheless, CLU is interesting as it provides evidence that there may be a fragment in our logic or in our specification language that is decidable, even for infinite state systems.

Much work on symbolic reachability analysis of infinite state systems has been undertaken, but most of it concentrates on safety properties only. Bouajjani et al. (see e.g. [6]) describe how *first-order arithmetical formulae*, expressing safety and liveness conditions, can be verified over Parametric Extended Automaton models, by specifying extra fairness conditions on the transitions of the models. The main difference with our approach is that we do not require fairness conditions on transitions of our models and that the first order modal μ-calculus is in fact capable of specifying fairness properties.

The technique by Bultan et al. [11] seems to be able to produce results that are comparable to ours. Their techniques, however, are entirely different from ours. In fact, their approach is similar to the approach used by Alur et al. [2] for hybrid systems. It uses affine constraints on integer variables, logical connectives and quantifiers to symbolically encode transition relations and sets of states. The logic, used to specify the properties

is a CTL-style logic. In order to guarantee termination, they introduce conservative approximation techniques that may yield "false negatives", which always converges. It is interesting to investigate whether the same conservative approximation techniques can be adapted to our techniques.

8.2. Summary

We discussed a pragmatic approach to verifying data-dependent systems. The techniques and procedure we presented, are based upon the techniques and algorithms, described by e.g. Mader [21]. Remark that, even though some of the theory we discussed in this paper was already investigated in slightly different settings, it was very clear that this approach could lead to effective tooling. The prototype tool implementation and the three sample verifications we discussed in this paper, show the approach is indeed viable and worthwhile.

Apart from the verifications, described in this paper, the prototype was successfully applied to other systems, see the discussion in [30, Chapter 3]. Summarising, we find that the verifications conducted with our prototype take in many cases an acceptable run-time. We expect improvements can still be made on the prototype. More importantly, we have been able to successfully use our prototype on systems with a finite (but extremely large) state-space, for which the standard μCRL tool-suite (which is competitive with other tool-suites that use explicit state-space representations) failed to calculate the exact state-space (see [30, Chapter 3]). Since this is where current state-of-the-art technologies break down, our technique is clearly a welcome addition.

Several other issues remain to be investigated. For instance, we think our technique may eventually be used to generalise specialised techniques, such as developed by Bryant et al. [10,27]. Also, in [17], we have identified rules and theorems for calculating with equation systems. These include special patterns and rules (such as the four deduction rules of Groote and Mateescu [12]) that would help skipping the (expensive) approximation step in our procedure. In the end, this is expected to improve the performance and efficacy of our tool.

References

[1] P. Abdulla, A. Bouajjani, B. Jonsson, On-the-fly analysis of systems with unbounded, lossy FIFO channels, in: A.J. Hu, M.Y. Vardi (Eds.), CAV'98, LNCS, vol. 1427, Springer-Verlag, 1998, pp. 305–318.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine, The algorithmic analysis of hybrid systems, Theoretical Computer Science 138 (1995) 3–34.

[3] J.C.M. Baeten, W.P. Weijland, Process Algebra, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

[4] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. Van Langevelde, B. Lisser, J.C. van de Pol, μCRL: A toolset for analysing algebraic specification, in: CAV'01, LNCS, vol. 2102, Springer-Verlag, 2001, pp. 250–254.

[5] B. Boigelot, P. Godefroid, B. Willems, P. Wolper, The power of QDDs, in: P. van Hentenryck (Ed.), Static Analysis, 4th International Symposium, SAS'97, LNCS, vol. 1302, Springer-Verlag, 1997, pp. 172–186.

[6] A. Bouajjani, A. Collomb-Annichini, Y. Lacknech, M. Sighireanu, Analysis of fair extended automata, in: P. Cousot (Ed.), Proceedings of SAS'01, LNCS, vol. 2126, Springer-Verlag, 2001, pp. 335–355.

[7] J.C. Bradfield, C. Stirling, Local model checking for infinite state spaces, Theoretical Computer Science 96 (1) (1992) 157–174.

[8] J.C. Bradfield, C. Stirling, Modal logics and mu-calculi, in: J.A. Bergstra, A. Ponse, S.A. Smolka (Eds.), Handbook of Process Algebra, Elsevier, North-Holland, 2001, pp. 293–330 (Chapter 4).
[9] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers C-35 (8) (1986) 677–691.
[10] R.E. Bryant, S.K. Lahiri, S.A. Seshia, Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions, in: CAV 2002, Lecture Notes in Computer Science, vol. 2404, Springer-Verlag, 2002, pp. 78–92.
[11] T. Bultan, R. Gerber, W. Pugh, Symbolic model checking of infinite state systems using Presburger arithmetic, in: O. Grumberg (Ed.), CAV'97, LNCS, vol. 1254, Springer-Verlag, 1997, pp. 400–411.
[12] J.F. Groote, R. Mateescu, Verification of temporal properties of processes in a setting with data, in: A.M. Haeberer (Ed.), AMAST'98, LNCS, vol. 1548, Springer-Verlag, 1999, pp. 74–90.
[13] J.F. Groote, A. Ponse, The syntax and semantics of μCRL, in: A. Ponse, C. Verhoef, S.F.M. van Vlijmen (Eds.), Algebra of Communicating Processes '94, Workshops in Computing Series, Springer-Verlag, 1995, pp. 26–62.
[14] J.F. Groote, A. Ponse, Y.S. Usenko, Linearization in parallel pCRL, Journal of Logic and Algebraic Programming 48 (1–2) (2001) 39–72.
[15] J.F. Groote, M.A. Reniers, Algebraic process verification, in: J.A. Bergstra, A. Ponse, S.A. Smolka (Eds.), Handbook of Process Algebra, Elsevier, North-Holland, 2001, pp. 1151–1208 (Chapter 17).
[16] J.F. Groote, J.C. van der Pol, Equational binary decision diagrams, in: M. Parigot, A. Voronkov (Eds.), LPAR2000, LNAI, vol. 1955, Springer-Verlag, 2000, pp. 161–178.
[17] J.F. Groote, T.A.C. Willemse, Parameterised boolean equation systems (extended abstract), in: Proceedings of CONCUR'04, 2004 (in press).
[18] D. Kozen, Results on the propositional mu-calculus, Theoretical Computer Science 27 (1983) 333–354.
[19] J.-L. Lassez, V.L. Nguyen, E.A. Sonenberg, Fixed point theorems and semantics: a folk tale, Information Processing Letters 14 (3) (1982) 112–116.
[20] S.P. Luttik, Choice quantification in process algebra, Ph.D. Thesis, University of Amsterdam, April, 2002.
[21] A. Mader, Verification of modal properties using Boolean Equation Systems, Ph.D. Thesis, Technical University of Munich, 1997.
[22] R. Mateescu, Local model-checking of an alternation-free value-based modal mu-calculus, in: A. Bossi, A. Cortesi, F. Levi (Eds.), Model Checking and Abstract Interpretation VMCAI'98, 1998.
[23] R. Mateescu, M. Sighireanu, Efficient on-the-fly model-checking for regular alternation-free mu-calculus, Science of Computer Programming 46 (3) (2003) 255–281.
[24] R. Milner, Communication and Concurrency, Prentice Hall International, 1989.
[25] A. Pnueli, J. Xu, L. Zuck, Liveness with (0, 1, infinity)-counter abstraction, in: E. Brinksma, K.G. Larsen (Eds.), CAV 2002, LNCS, vol. 2404, Springer-Verlag, 2002, pp. 107–122.
[26] M. Raynal, Algorithms for Mutual Exclusion, North Oxford Academic, 1986.
[27] O. Strichman, S.A. Seshia, R.E. Bryant, Deciding separation formulas with SAT, in: CAV 2002, LNCS, vol. 2404, Springer-Verlag, 2002, pp. 209–222.
[28] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5 (1955) 285–309.
[29] Y.S. Usenko, Linearization in μCRL. Ph.D. Thesis, Eindhoven University of Technology, December, 2002.
[30] T.A.C. Willemse, Semantics and Verification in process algebras with data and timing, Ph.D. Thesis, Eindhoven University of Technology, February, 2003.